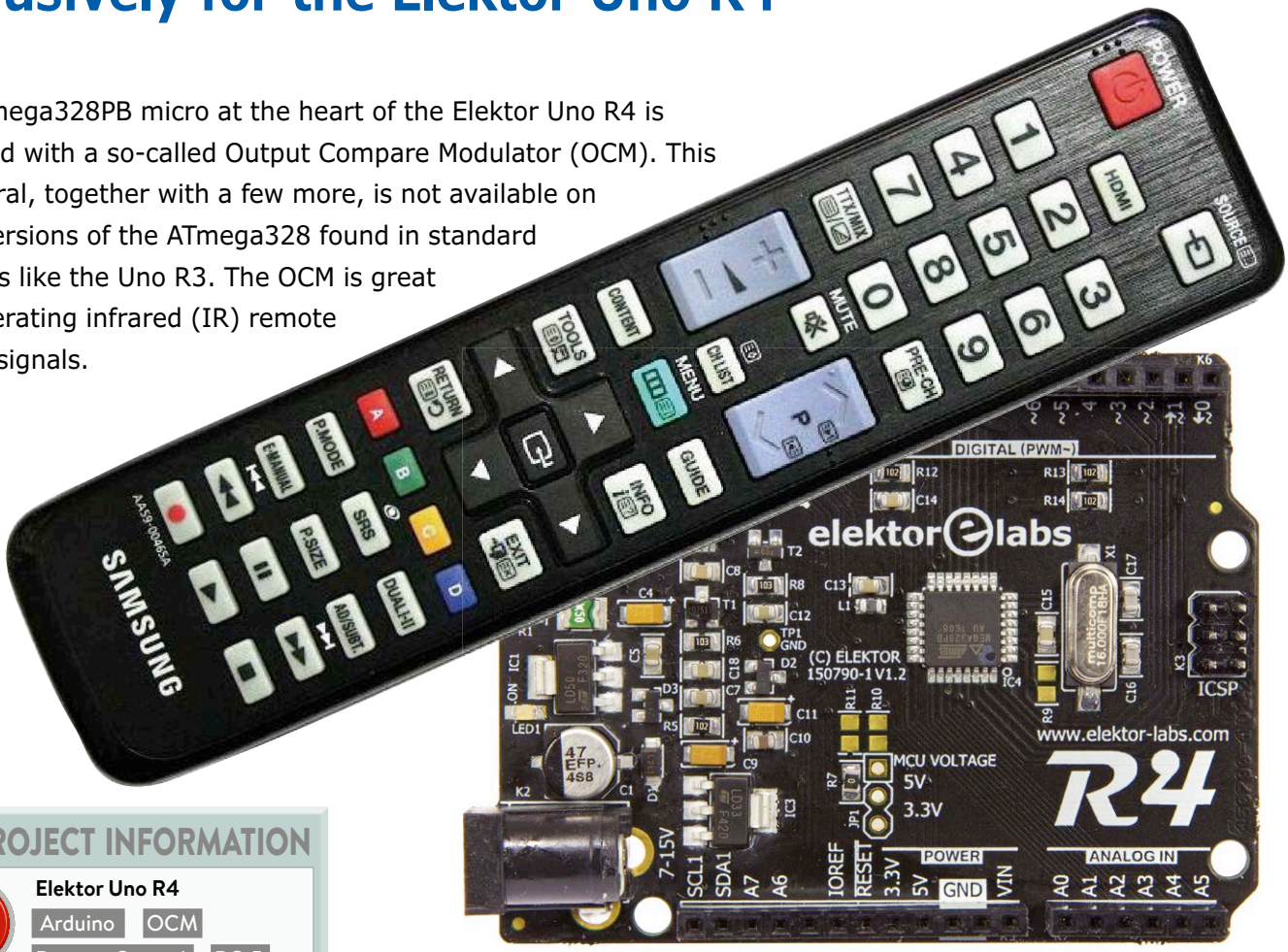
How to Produce NEC-style Remote Control Signals

Exclusively for the Elektor Uno R4

The ATmega328PB micro at the heart of the Elektor Uno R4 is equipped with a so-called Output Compare Modulator (OCM). This peripheral, together with a few more, is not available on other versions of the ATmega328 found in standard Arduinos like the Uno R3. The OCM is great for generating infrared (IR) remote control signals.



PROJECT INFORMATION

Elektor Uno R4

Arduino | OCM | Remote Control | RC-5

entry level
➡ intermediate level
expert level

1 hour approximately

Elektor Uno R4 computer with Arduino IDE

€25 / $30 / £20 approximately

By **Clemens Valens** (Elektor Labs)

The OCM combines the outputs of Timer3 and Timer4 — another Elektor Uno R4 exclusivity — and makes the result available on PD2, which is Arduino pin 2 and MCU pin 32. Here, combining means either AND-ing or OR-ing the two signals, as shown in **Figure 1**. The value of register PORTD2 determines which operation is executed. If PORTD2 is set to '1', then OR-mode is selected; if PORTD2 is set to '0', AND-mode is what you get. Signals similar to those from Figure 1 are commonly found in infrared remote controls where they are used to modulate the current through an IR LED.

My home, and yours too I reckon, is littered with remote controls (RCs). Some time ago when I needed a Philips RC-5-compatible RC I had to try them all only to discover that none of them used the RC-5 protocol, not even my Philips DVD player (using RC-6). Most of my RCs appeared to use some variant of the Japanese NEC protocol instead. This is understandable as most of the remote-controlled equipment in my house, and yours too I reckon, was made

in Asia. So, now that we have this nice OCM peripheral, let's use it to produce NEC-compatible RC signals.
An NEC-type RC code starts with a preamble or burst of 9 ms, followed by 4.5 ms of silence (a pause). Next the bits, 32 in total, are transmitted one after the other. A '0' consists of a burst and a pause of both 562.5 µs (1,125 µs in total) and a '1' is a burst of 562.5 µs with a pause of 1,687.5 µs (2,250 µs in total). The sequence must be terminated by a closing burst (postamble) to 'close' the last bit. All durations mentioned here are multiples of 562.5 µs. In what follows, a 'pulse' is the duration of a burst, and a 'period' is the duration of a burst + pause.
A burst is nothing more than a high-frequency signal, a carrier, switched on and off by the modulator. We can use Timer3 to generate a carrier and modulate it with Timer4. The other way around is equally valid, but in life one is forced to make choices. All timers in the ATmega328, including Timer3 and Timer4, are capable of generating interrupts when they overflow or when they exceed a certain value. This offers a nice mechanism to easily modulate the carrier with the data we wish to transmit: every time the interrupt fires, the next bit to transmit is read from the TX buffer and used to determine the pulse and period needed to send a zero or a one. The sketch that follows shows how we can accomplish this.

```
/*
 * IR Transmitter
 * Uses Output Compare Module (OCM), ATmega328PB
   only.
 */

#define CARRIER  38000
#define RC_CODE  0x6170807f

const int ir_led = 2;
const uint16_t pulse = 1125; // 2*562.5
uint32_t tx_buffer = 0;
uint32_t tx_index = 0;

#define TIMER3_ON  TCCR3B |= (1<<CS30) /* prescaler
   1 */
#define TIMER3_OFF  TCCR3B &= ~(1<<CS30)
#define TIMER4_ON  TCCR4B |= (1<<CS41) /* prescaler
   8 */
#define TIMER4_OFF  TCCR4B &= ~(1<<CS41)

ISR(TIMER4_COMPB_vect)
{
  if (tx_index!=0) // Any bits left?
  {
    OCR4B = pulse; // Set pulse length.
    if ((tx_buffer&tx_index)!=0)
    {
      OCR4A = OCR4B<<2; // Set '1' period.
    }
    else
    {
      OCR4A = OCR4B<<1; // Set '0' period.
    }
    tx_index >>= 1; // Next bit.
  }
  else
  {
    if (OCR4A>=2*OCR4B)
    {
      OCR4A = OCR4B + 10; // Send closing burst.
    }
    else
    {
      // Done, clean up.
      TIMSK4 &= ~(1<<OCIE4B); // Disable interrupts.
      TIMER3_OFF;
      TIMER4_OFF;
    }
  }
}

void ir_send_code(uint32_t code)
{
  // Copy code to TX buffer.
  tx_buffer = code;
  tx_index = 0x80000000; // 32 bits.
  // Setup timer4 for preamble.
  OCR4A = 2*12*pulse;
  OCR4B = 2*8*pulse;
  TIFR4 = (1<<OCF4B); // Clear pending interrupts.
  TIMSK4 = (1<<OCIE4B); // Enable compare interrupt.
  // Go!
  TIMER3_ON;
  TIMER4_ON;
}

void setup(void)
{
  pinMode(ir_led,OUTPUT); // Enable output.
  digitalWrite(ir_led,LOW); // OCM in AND mode.
  // Timer3 mode 4 (CTC), toggle output on match.
  TCCR3A = (1<<COM3B0);
  TCCR3B = (1<<WGM32);
  OCR3A = 16000000/CARRIER/2; // Toggle at 76 kHz ->
   38 kHz.
  OCR3B = OCR3A>>1; // 50% duty-cycle.
  // Timer4 mode 15 (Fast PWM), clear output on
   match.
  TCCR4A = (1<<WGM41) | (1<<WGM40) | (1<<COM4B1);
  TCCR4B = (1<<WGM43) | (1<<WGM42);
  ir_send_code(RC_CODE);
}

void loop(void)
{
}
```

It all starts in the function `setup` with the declaration of pin 2 (PD2, to which the IR LED is to be connected) as an output. Skip this step and nothing will ever come out of this pin. Making the

output low not only switches off the LED, it also puts the OCM in AND mode. Next we set up Timer3 in CTC (compare) mode 4 for a frequency of 38 kHz, a popular RC carrier frequency. Note how the timer seems to be set up for twice this frequency, but the factor of two is necessary to compensate for the toggling of the output pin on every compare match, effectively dividing the signal frequency by two and resulting in a 38-kHz signal at the output. Pin 2 is the OC3B output controlled by OCR3B, consequently we set this register to half the value of OCR3A to obtain a duty-cycle of 50%. Timer4 is configured in Fast PWM mode 14 where OCR4A determines the frequency and OCR4B the duty-cycle. The rest of its configuration is delayed till the function `send_code`.

This function starts by copying the code to transmit into the 32-bit TX buffer. Because a code starts with a preamble, the corresponding pulse (9 ms) and period (13.5 ms) values are loaded into the pulse (OCR4B) and period (OCR4A) registers. Here a multiplication by two is needed because the pulse and period are specified in microseconds, but since the MCU clock is 16 MHz and the prescaler can only divide it by 1, 8, 64, 256 or 1024, not by 16, we are forced to use 8 instead and we're off by a factor of two. The next step is to prepare for interrupts and start the timers.

The COMPB interrupt service routine is called at the end of a pulse. When this happens, and all the bits haven't been sent yet, the timer needs to be reconfigured for a period of either 1,125 µs (if the bit to transmit is a '0') or 2,250 µs (if the bit to transmit is a '1'), and set the pulse length to 562.5 µs (identical for '0's and '1's). As before, all the values are multiplied by two to compensate for the prescaler value of 8 which we would have liked to be 16. The 32-bit code must be terminated by a closing burst, otherwise the receiver can't tell if the last bit is a '0' or a '1'. When `tx_index` equals zero we know that all bits have been sent. We can now use the OCR4A register, for instance, with a special value to send a final burst, meaning that the ISR will run one more time. If we choose a special value smaller than 1,125 µs (but larger than OCR4B), this state will be easy to detect by the ISR the next time it is called. Naturally it is possible to use another mechanism with a static variable or something similar, but the register is available, so why not use it? The last thing the ISR does after the preamble, the 32 code bits and the stop burst have all been sent, is switch off the lights, ermm, the timers as they are no longer needed.
The function `loop` has remained empty because we send the RC code only once after pressing the Reset button.

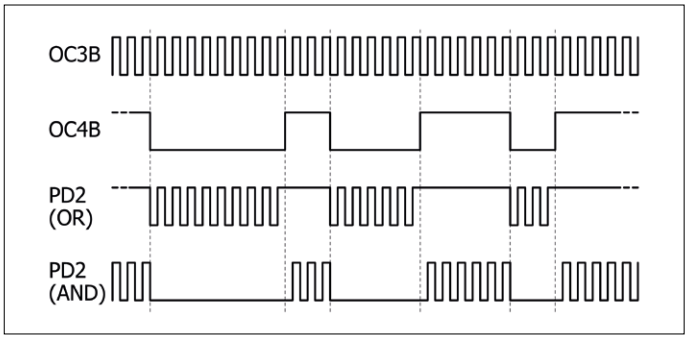


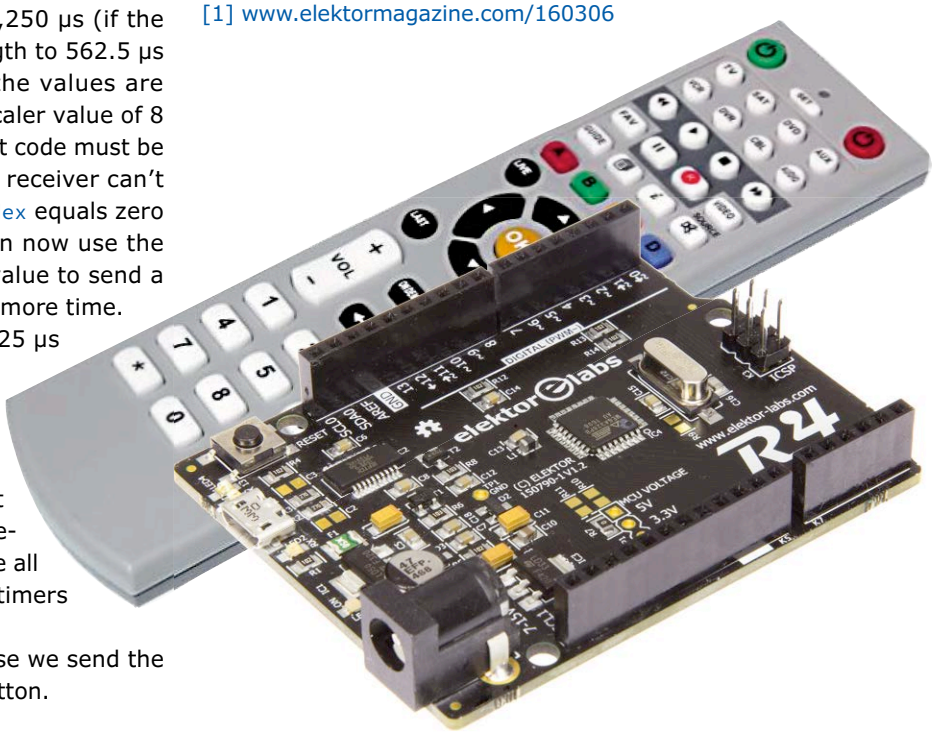
Figure 1. In OCM mode PD2 outputs either OC3B OR OC4B (PORTD2 = 1) or OC3B AND OC4B (PORTD2 = 0).

Transmitting a code is fully interrupt-driven and the rest of the program can do other things while this runs in the background. The program can poll the OCIE4B bit in the TIMSK4 register to check if the transmitter is still busy. Furthermore, as you can see, there is no interaction whatsoever between the timers 3 and 4 except for the ISR switching them both off at the end (which is optional), and there are no pins to be set or cleared. Everything is handled directly by the hardware. The downside of this simplicity is that it works on pin 2 only. ⏮

(160306)

Web Link

[1] www.elektormagazine.com/160306



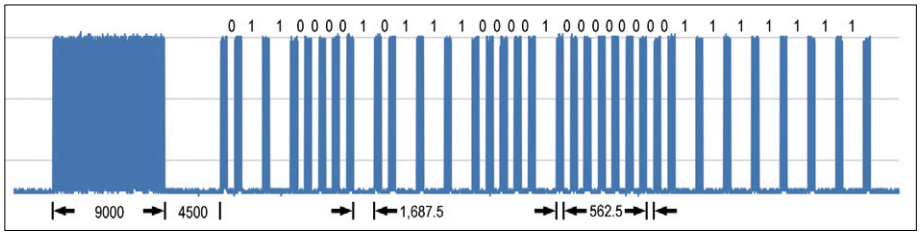


Figure 2. A scope plot of the signal on PD2 produced by the sketch.